# The l3build package
# Checking and building packages*

The LaTeX3 Project†

Released 2016/01/19

## Contents


**1 The l3build system** **1**
- 1.1 Introduction . . . . . . . . . 1
- 1.2 Main build commands . . . . 2
- 1.3 Example build scripts . . . . 7
- 1.4 Variables . . . . . . . . . . . 7
- 1.5 Dependencies . . . . . . . . . 11
- 1.6 Output normalisation . . . . 11

**2 Writing test files** **13**
- 2.1 Metadata and structural commands . . . . . . . . . . 13
- 2.2 Commands to help write tests 13
- 2.3 Showing box content . . . . . 15

**3 Alternative test formats** **15**
- 3.1 Generating test files with DocStrip . . . . . . . . . . . . 15
- 3.2 Specifying expectations . . . 17

**4 Release-focussed features** **17**
- 4.1 Automatic version modification . . . . . . . . . . . . . . 17
- 4.2 Typesetting documentation . 18


# 1 The l3build system

## 1.1 Introduction

The l3build system is a Lua script for building TeX packages, with particular emphasis on regression testing. It is written in cross-platform Lua code, so can be used by any modern TeX distribution with the `texlua` interpreter. A package for building with l3build can be written in any TeX dialect; its defaults are set up for LaTeX packages written in the DocStrip style. (Caveat: minimal testing has yet been performed for non-LaTeX packages.)

Test files are written as standalone TeX documents using the `regression-test.tex` setup file; documentation on writing these tests is discussed in Section 2.

The `l3build.lua` script is not designed to be executed directly; each package will define its own `build.lua` script as a driver file which both sets variables (such as the name of the package) and then calls the main `l3build.lua` script internally.

*This file describes v6377, last revised 2016/01/19.

†E-mail: latex-team@latex-project.org

A standard package layout might look something like the following:

```
abc/
    abc.dtx
    abc.ins
    build.lua
    README.md
    support/
    testfiles/
```

Most of this should look fairly self-explanatory. The top level `support/` directory (optional) would contain any necessary files for compiling documentation, running regression tests, and so on.

The l3build system is also capable of building and checking *bundles* of packages. To avoid confusion, we refer to either a standalone package or a package within a bundle as a *module*.

For example, within the LaTeX3 project we have the l3packages bundle which contains the xparse, xtemplate, etc., modules. These are all built and distributed as one bundle for installation, distribution *via* CTAN and so forth.

Each module in a bundle will have its own build script, and a bundle build script brings them all together. A standard bundle layout would contain the following structure.

```
mybundle/
         build.lua
         support/
         yyy/                    zoo/
             build.lua               build.lua
             README.md               README.md
             testfiles/              testfiles/
             yyy.dtx                 zoo.dtx
             yyy.ins                 zoo.ins
```

All modules within a bundle must use the same build script name.

In a small number of cases, the name used by CTAN for a module or bundle is different from that used in the installation tree. For example, the LaTeX $2_\varepsilon$ kernel is called latex-base by CTAN but is located inside ⟨*texmf*⟩`/tex/latex/base`. This can be handled by using `ctanpkg` for the name required by CTAN to override the standard value.

The `testfiles/` folder is local to each module, and its layout consists of a series of regression tests with their outputs.

```
testfiles/
          test1.lvt
          test1.tlg
          ...
          support/
                  my-test.cls
```

Again, the `support/` directory contains any files necessary to run some or all of these tests.

When the build system runs, it creates a directory `build/` for various unpacking, compilation, and testing purposes. For a module, this build folder can be in the main directory of the package itself, but for a bundle it should be common for the bundle itself and for all modules within that bundle. A `build/` folder can be safety deleted; all material within is re-generated for each command of the l3build system.

## 1.2 Main build commands

In the working directory of a bundle or module, the following commands can be executed:

- `check`
- `check` ⟨*name(s)*⟩
- `cmdcheck`
- `clean`
- `doc`
- `install`
- `save` ⟨*name(s)*⟩

These commands are described below.

As well as these commands, the system recognises the options

- `-engine` (`-e`) Sets the engine to use for testing
- `-halt-on-error` (`-H`) Specifies that checks should stop as soon as possible, rather than running all requested tests
- `-quiet` (`-q`) Suppresses output from unpacking

```
$ texlua build.lua check
```

The `check` command runs the entire test suite. This involves iterating through each `.lvt` file in the test directory (specified by the `testfiledir` variable), compiling each test in a "sandbox" (a directory specified by `testdir`), and comparing the output against each matching predefined `.tlg` file.

If changes to the package or the typesetting environment have affected the results, the check for that file fails. A `diff` of the expected to actual output should then be inspected to determine the cause of the error; it is located in the `testdir` directory (default `maindir .. "/build/test"`).

On Windows, the `diff` program is not available and so `fc` is used instead (generating an `.fc` file). Setting the environmental variables `diffexe` and `diffext` can be used to adjust the choice of comparison made: the standard values are

Windows `diffext = fc`, `diffexe = fc /n`

*nix `diffext = diff`, `diffexe = diff -c --strip-trailing-cr`

The following files are moved into the "sandbox" for the `check` process:

- all `installfiles` after unpacking;
- all `checkfiles` after unpacking;
- any files in the directory `testsuppdir`;
- any files that match `checksuppfiles` in the `supportdir`.

This range of possibilities allow sensible defaults but significant flexibility for defining your own test setups.

Checking can be performed with any or all of the 'engines' `pdftex`, `xetex`, and `luatex`. By default, each test is executed with all three, being compared against the `.tlg` file produced from the `pdftex` engine (these defaults are controlled by the `checkengines` and `stdengine` variable respectively). The format used for tests can be altered by setting `checkformat`: the default setting `latex` means that tests are run using *e.g.* `pdflatex`, whereas setting to `plain` will run tests using *e.g.* `pdftex`. (Currently, this should be one of `latex` or `plain`.) To perform the check, the engine typesets each test `checkruns` times. More detail on this in the documentation on `save`. Options passed to the binary are defined in the variable `checkopts`.

By default, `texmf` trees are searched for input files when checking. This can be disabled by setting `checksearch` to `false`: isolation provides confidence that the tests cannot accidentally be running with incorrect files installed in the main distribution or `hometexmf`.

`$ texlua build.lua check ⟨name(s)⟩`

Checks only the test `⟨name(s)⟩.lvt`. All engines specified by `checkengines` are tested unless the command line option `-engine` (or `-e`) has been given to limit testing to a single engine.

`$ texlua build.lua cmdcheck`

For l3doc-based sources, allows checking that the commands defined in the code part (by `cmdchkfiles`) are documented in the description part. This is performed by passing the `check` option to the l3doc class, typesetting the file(s) to check with engine `stdengine` with options `cmdchkopts`, and checking the resultant `.cmds` file(s). Dependencies are specified also with `checkdeps`.

`$ texlua build.lua clean`

This command removes all temporary files used for package bundling and regression testing. In the standard layout, these are all files within the directories defined by `localdir`, `testdir`, `typesetdir` and `unpackdir`, as well as all files defined in the `cleanfiles` variable in the same directory as the script. The defaults are `.pdf` files from typesetting (`doc`) and `.zip` files from bundling (`ctan`).

`$ texlua build.lua ctan`

Creates an archive of the package and its documentation, suitable for uploading to CTAN The archive is compiled in `distribdir`, and if the results are successful the resultant `.zip` file is moved into the same directory as the build script. If `packtdszip` is set true then the building process includes a `.tds.zip` file containing the 'TeX Directory Structure' layout of the package or bundle. The archive therefore may contain two 'views' of the package:

```
abc.zip/
        abc/
            abc.dtx
            abc.ins
            abc.pdf
            README.md
        abc.tds.zip/
                    doc/latex/abc/
                                  abc.pdf
                                  README.md
                 source/latex/abc/
                                  abc.dtx
                                  abc.ins
                    tex/latex/abc/
                                  abc.sty
```

The files copied into the archive are controlled by a number of variables. The 'root' of the TDS structure is defined by `tdsroot`, which is `"latex"` by default. Plain users would redefine this to `"plain"` (or perhaps `"generic"`), for example. The build process for a `.tds.zip` file currently assumes a 'standard' structure in which all extracted files should be placed inside the `tex` tree in a single directory, as shown above. If the module includes any BibTeX or MakeIndex styles these will be placed in the appropriate subtrees.

The `doc` tree is constructed from:

- all files matched by `demofiles`,
- all files matched by `docfiles`,
- all files matched by `typesetfiles` with their extension replaced with `.pdf`,
- all files matched by `textfiles`,
- all files matched by `bibfiles`.

The `source` tree is constructed from all files matched by `typesetfiles` and `sourcefiles`. The `tex` tree from all files matched by `installfiles`.

Files that should always be excluded from the archive are matched against the `excludefiles` variable; by default this is `{"*~"}`, which match Emacs' autosave files.

Binary files should be specified with the `binaryfiles` variable (default `{"*.pdf", "*.zip"}`); these are added to the zip archive without normalising line endings (text files are automatically converted to Unix-style line endings).

To create the archive, by default the binary `zipexe` is used (`"zip"`) with options `zipopts` (`-v -r -X`). The intermediate build directories `ctandir` and `tdsdir` are used to construct the archive.

`$ texlua build.lua doc`

Compiles documentation files in the `typesetdir` directory. In the absence of one or more file names, all documentation is typeset; a file list may be given at the command line for selective typesetting. If the compilation is successful the `.pdf` is moved back into the main directory.

The documentation compilation is performed with the `typesetexe` binary (default `pdflatex`), with options `typesetopts`. Additional TeX material defined in `typesetcmds` is passed to the document (e.g., for writing `\\PassOptionsToClass{l3doc}{letterpaper}`, and so on—note that backslashes need to be escaped in Lua strings).

Files that match `typesetsuppfiles` in the `support` directory (`supportdir`) are copied into the `build/local` directory (`localdir`) for the typesetting compilation process. Additional dependencies listed in the `typesetdeps` variable (empty by default) will also be installed.

If `typesetsearch` is `true` (default), standard `texmf` search trees are used in the typesetting compilation. If set to false, *all* necessary files for compilation must be included in the `build/local` sandbox.

`$ texlua build.lua install`

Copies all package files (defined by `installfiles`) into the user's home `texmf` tree in the form of the TeX Directory Structure.

`$ texlua build.lua save ⟨name(s)⟩`

This command runs through the same execution as `check` for a specific test(s) ⟨*name(s)*⟩`.lvt`. This command saves the output of the test to a `.tlg` file. This file is then used in all subsequent checks against the ⟨*name*⟩`.lvt` test.

If the `-engine` (or `-e`) is specified (one of `pdftex`, `xetex`, or `luatex`), the saved output is stored in ⟨*name*⟩`.`⟨*engine*⟩`.tlg`. This is necessary if running the test through a different engine produces a different output. A normalisation process is performed when checking to avoid common differences such as register allocation; full details are listed in section 1.6.

`$ texlua build.lua setversion`

Modifies the content of files specified by `versionfiles` to allow automatic updating of the file date and version. The latter are specified using the `-d` and `-v` command line options and if not given will default to the current date in ISO format (YYYY-MM-DD) and `-1`, respectively. As detailed below, the standard set up has no search pattern defined for this target and so no action will be taken *unless* a version type for substitution is set up (using `versionform` or by defining a custom function).

`$ texlua build.lua unpack`

This is an internal target that is normally not needed on user level. It unpacks all files into the directory defined by `unpackdir`. This occurs before other build commands such as `doc`, `check`, etc.

The unpacking process is performed by executing the `unpackexe` (default `tex`) with options `unpackopts` on all files defined by the `unpackfiles` variable; by default, all files that match `{"*.ins"}`.

If additional support files are required for the unpacking process, these can be enumerated in the `unpacksuppfiles` variable. Dependencies for unpacking are defined with `unpackdeps`.

By default this process allows files to be accessed in all standard `texmf` trees; this can be disabled by setting `unpacksearch` to `false`.

## 1.3 Example build scripts

An example of a standalone build script for a package that uses self-contained `.dtx` files is shown in Figure 1. Here, the `module` only is defined, and since it doesn't use `.ins` files so the variable `unpackfiles` is redefined to run `tex` on the `.dtx` files instead to generate the necessary `.sty` files. There are some PDFs in the repository that shouldn't be part of a CTAN submission, so they're explicitly excluded, and here unpacking is done 'quietly' to minimise console output when building the package. Finally, because this is a standalone package, we assume that l3build is installed in the main TeX distribution and find the Lua script by searching for it.

An example of a bundle build script for l3packages is shown in Figure 2. Note for LaTeX3 we use a common file to set all build variables in one place, and the path to the `l3build.lua` script is hard-coded so we always use our own most recent version of the script. An example of an accompanying module build script is shown in Figure 3.

Under a Unix-like platform, you may wish to run '`chmod +x build.lua`' on these files, which allows a simpler command line use. Instead of writing

```
texlua build.lua check
```

for example, you would simply write

```
./build.lua check
```

```
#!/usr/bin/env texlua

-- Build script for breqn

module = "breqn"

unpackfiles = {"*.dtx"}
excludefiles = {"*/breqn-abbr-test.pdf",
                "*/eqbreaks.pdf"}
unpackopts  = "-interaction=batchmode"

kpse.set_program_name("kpsewhich")
dofile(kpse.lookup("l3build.lua"))
```

Figure 1: The build script for the breqn package.

```
#!/usr/bin/env texlua

-- Build script for LaTeX3 "l3packages" files

-- Identify the bundle: there is no module as this is the "driver"
bundle = "l3packages"

-- Location of main directory: use Unix-style path separators
maindir = ".."

-- Load the common build code: this is the one place that a path
-- needs to be hard-coded
dofile (maindir .. "/l3build/l3build-config.lua")
dofile (maindir .. "/l3build/l3build.lua")
```

Figure 2: The build script for the l3packages bundle.

```
#!/usr/bin/env texlua

-- Build script for LaTeX3 "xparse" files

-- Identify the bundle and module:
bundle = "l3packages"
module = "xparse"

-- Location of main directory: use Unix-style path separators
-- Should match that defined by the bundle.
maindir = "../.."

-- Load the common build code: this is the one place that a path
-- needs to be hard-coded
dofile (maindir .. "/l3build/l3build-config.lua")
dofile (maindir .. "/l3build/l3build.lua")
```

Figure 3: The build script for the xparse module.

```
@echo off
texlua build.lua %*
```

Figure 4: Windows batch file wrapper for running the build process.

instead. (Or even omit the `./` depending on your path settings.) Windows users can achieve a similar effect by creating a file `build.bat` as show in Figure 4.

### 1.4 Variables

This section lists all variables defined in the `l3build.lua` script that are available for customisation.

| Variable | Default | Description |
|---|---|---|
| `module` | `""` | The name of the module. |
| `bundle` | `""` | The name of the bundle in which the module belongs. |
| `ctanpkg` | `bundle` | Name of the bundle on CTAN |
| `modules` | `{ }` | The list of all modules in a bundle (when not auto-detecting) |
| `exclmodules` | `{ }` | Directories to be excluded from automatic module detection |
| `maindir` | `"."` | The top level directory for this module or bundle. |
| `supportdir` | `maindir .. "/support"` | Where copies of files to support check/doc compilation are stored. |
| `testfiledir` | `maindir .. "/testfiles"` | Where the tests are. |
| `testsuppdir` | `testfiledir .. "/support"` | Where support files for the tests are. |
| `localdir` | `maindir .. "/build/local"` | Generated folder where support files are placed to allow "sandboxed" TeX runs. |
| `testdir` | `maindir .. "/build/test"` | Generated folder where tests are run. |
| `typesetdir` | `maindir .. "/build/doc"` | Generated folder where typesetting is run. |
| `unpackdir` | `maindir .. "/build/unpack"` | Generated folder where unpacking occurs. |
| `distribdir` | `maindir .. "/build/distrib"` | Generated folder where the archive is created. |
| `ctandir` | `distribdir .. "/ctan"` | Generated folder where files are organised for CTAN. |
| `tdsdir` | `distribdir .. "/tds"` | Generated folder where files are organised for a TDS. |
| `tdsroot` | `"latex"` | Root directory of the TDS structure for the bundle/module to be installed into. |
| `bibfiles` | `{"*.bst"}` | BibTeX database files. |
| `binaryfiles` | `{"*.pdf", "*.zip"}` | Files to be added in binary mode to zip files. |
| `bstfiles` | `{"*.bst"}` | BibTeX style files. |
| `checkfiles` | `{ }` | Extra files unpacked purely for tests |
| `checksuppfiles` | | Files needed for performing regression tests. |
| `cmdchkfiles` | `{ }` | Files need to perform command checking (l3doc-based documentation only). |
| `cleanfiles` | `{"*.log", "*.pdf", "*.zip"}` | Files to delete when cleaning. |
| `demofiles` | `{ }` | Files which show how to use a module. |
| `docfiles` | `{ }` | Files which are part of the documentation but should not be typeset. |
| `excludefiles` | `{"*~"}` | Files to ignore entirely (default for Emacs backup files). |
| `installfiles` | `{"*.sty"}` | Files to install to the TeX tree and similar tasks. |
| `makeindexfiles` | `{"*.ist"}` | MakeIndex files to be included in a TDS-style zip |

| Variable | Default | Description |
|---|---|---|
| `sourcefiles` | `{"*.dtx", "*.ins"}` | Files to copy for unpacking. |
| `textfiles` | `{"*.md", "*.txt"}` | Plain text files to send to CTAN as-is. |
| `typesetfiles` | `{"*.dtx"}` | Files to typeset for documentation. |
| `typesetsuppfiles` | `{ }` | Files needed to support typesetting when "sandboxed". |
| `unpackfiles` | `{"*.ins"}` | Files to run to perform unpacking. |
| `unpacksuppfiles` | `{ }` | Files needed to support unpacking when "sandboxed". |
| `versionfiles` | `{"*.dtx"}` | Files for automatic version editing. |
| `bakext` | `".bak"` | Extension of backup files. |
| `lvtext` | `".lvt"` | Extension of test files. |
| `tlgext` | `".tlg"` | Extension of test file output. |
| `lvtext` | `".lve"` | Extension of auto-generating test file output. |
| `logext` | `".log"` | Extension of checking output, before processing it into a `.tlg`. |
| `checkdeps` | `{ }` | List of build unpack dependencies for checking. |
| `typesetdeps` | `{ }` | ... for typesetting docs. |
| `unpackdeps` | `{ }` | ... for unpacking. |
| `checkengines` | `{"pdftex", "xetex", "luatex"}` | Engines to check with `check` by default. |
| `stdengine` | `"pdtex"` | Engine to generate `.tlg` file from. |
| `checkformat` | `"latex"` | Format to use for tests. |
| `typesetexe` | `"pdflatex"` | Executable for compiling `doc(s)`. |
| `unpackexe` | `"tex"` | Executable for running `unpack`. |
| `zipexe` | `"zip"` | Executable for creating archive with `ctan`. |
| `checkopts` | `"-interaction=batchmode"` | Options based to engine when running checks. |
| `cmdchkopts` | `"-interaction=batchmode"` | Options based to engine when running command checks. |
| `typesetopts` | `"-interaction=nonstopmode"` | Options based to engine when typesetting. |
| `unpackopts` | `""` | Options based to engine when unpacking. |
| `zipopts` | `"-v -r -X"` | Options based to zip program. |
| `checksearch` | `true` | Look in `tds` dirs for checking? |
| `typesetsearch` | `true` | Look in `tds` dirs for typesetting docs? |
| `unpacksearch` | `true` | Look in `tds` dirs for unpacking? |
| `glossarystyle` | `"gglo.ist"` | MakeIndex style file for glossary/changes creation |
| `indexstyle` | `"gind.ist"` | MakeIndex style for index creation |
| `biberexe` | `"biber"` | Biber executable |
| `biberopts` | `""` | Biber options |
| `bibtexexe` | `"bibtex8"` | BibTeX executable |
| `bibtexopts` | `"-W"` | BibTeX options |
| `makeindexexe` | `"makeindex"` | MakeIndex executable |
| `makeindexopts` | `""` | MakeIndex options |
| `asciiengines` | `{"pdftex"}` | Engines which should log as sure ASCII |

| Variable | Default | Description |
|---|---|---|
| `checkruns` | `1` | How many times to run a check file before comparing the log. |
| `maxprintline` | `79` | Length of line to use in log files. |
| `packtdszip` | `false` | Build a TDS-style zip file for CTAN? |
| `scriptname` | `"build.lua"` | Name of script used in dependencies. |
| `typesetcmds` | `""` | Instructions to be passed to TeX when doing typesetting. |
| `versionform` | `""` | Nature of version strings for auto-replacement. |

### 1.5 Dependencies

If you have multiple packages that are developed separately but still interact in some way, it's often desirable to integrate them when performing regression tests. For LaTeX3, for example, when we make changes to l3kernel it's important to check that the tests for l3packages still run correctly, so it's necessary to include the l3kernel files in the build process for l3packages.

In other words, l3packages is *dependent* on l3kernel, and this is specified in l3build by setting appropriately the variables `checkdeps`, `typesetdeps`, and `unpackdeps`. The relevant parts of the LaTeX3 repository is structured as the following.

```
l3/
   l3kernel/
              build.lua
              expl3.dtx
              expl3.ins
              ...
              testfiles/
   l3packages/
              build.lua
              xparse/
                     build.lua
                     testfiles/
                     xparse.dtx
                     xparse.ins
   support/
```

For LaTeX3 build files, `maindir` is defined as top level folder `l3`, so all support files are located here, and the build directories will be created there. To set l3kernel as a dependency of l3package, within `l3packages/xparse/build.lua` the equivalent of the following is set:

```
maindir = "../.."
checkdeps = {maindir .. "/l3kernel"}
```

This ensures that the l3kernel code is included in all processes involved in unpacking and checking and so on. The name of the script file in the dependency is set with the `scriptname` variable; by default these are `"build.lua"`.

## 1.6 Output normalisation

To allow test files to be used between different systems (*e.g.* when multiple developers are involved in a project), the log files are normalised before comparison during checking. This removes some system-dependent data but also some variations due to different engines. This normalisation consists of two parts: removing ("ignoring") some lines and modifying others to give consistent test. Currently, the following types of line are ignored:

- Lines before the `\START`, after the `\END` and within `\OMIT`/`\TIMO` blocks
- Entirely blank lines, including those consisting only of spaces.
- Lines containing file dates in format ⟨*yyyy*⟩/⟨*mm*⟩/⟨*dd*⟩.
- Lines starting `\openin` or `\openout`.

Modifications made in lines are:

- Removal of the name of the test file itself.
- Removal of the `pdftex.map` load information given during first page shipout.
- Removal spaces at the start of lines.
- Removal of `./` at start of file names.
- Standardisation of the list of units known to TeX (pdfTeX and LuaTeX add a small number of additional units which are not known to TeX90 or XeTeX).
- Standardisation of `\csname\endcsname␣` to `\csname\endcsname` (the former is formally correct, but the latter was produced for many years due to a TeX bug).
- Conversion of `on line` ⟨*number*⟩ to `on line ...` to allow flexibility in changes to test files.
- 

LuaTeX makes several additional changes to the log file. As normalising these may not be desirable in all cases, they are handled separately. When creating LuaTeX-specific test files (either with LuaTeX as the standard engine or saving a LuaTeX-specific `.tlg` file) no further normalisation is undertaken. On the other hand, for cross-engine comparison the following normalisation is applied:

- Removal of additional (unused) `\discretionary` points.
- Removal of `U+...` notation for missing characters.
- Removal of `display` for display math boxes (included by TeX90/pdfTeX/XeTeX).
- Removal of Omega-like `direction TLT` information.
- Removal of additional whatsit containing local paragraph information (`\localinterlinepenalty`, *etc.*).

- Rounding of glue set to four decimal places (glue set may be slightly different in LuaTeX compared to other engines).
- Conversion of low chars (1 to 31) to `^^` notation.

When making comparisons between 8-bit and Unicode engines it is useful to format the top half of the 8-bit range such that it appears in the log as `^^`⟨*char*⟩ (the exact nature of the 8-bit output is otherwise dependent on the active code page). This may be controlled using the `asciiengines` option. Any engines named here will use a `.tcx` file to produce only ASCII chars in the log output, whilst for other engines normalisation is carried out from UTF-8 to ASCII. If the option is set to an empty table the latter process is skipped: suitable for cases where only Unicode engines are in use.

# 2 Writing test files

Test files are written in a TeX dialect using the support file `regression-test.tex`, which should be `\input` at the very beginning of each test. Additional customisations to this driver can be included in a local `regression-test.cfg` file, which will be loaded automatically if found.

The macros loaded by `regression-test.tex` set up the test system and provide a number of commands to aid the production of a structured test suite. The basis of the test suite is to output material into the `.log` file, from which a normalised test output (`.tlg`) file is produced by the build command `save`. A number of commands are provided for this; they are all written in uppercase to help avoid possible conflicts with other package commands.

## 2.1 Metadata and structural commands

Any commands that write content to the `.log` file that should be ignored can be surrounded by `\OMIT` ... `\TIMO`. At the appropriate location in the document where the `.log` comparisons should start (say, after `\begin{document}`), the test suite must contain the `\START` macro. The test should then include `\AUTHOR{`⟨*authors details*⟩`}` in case a test file fails in the future and needs to be re-analysed.

Some additional diagnostic information can then be included as metadata for the conditions of the test. The desired format can be indicated with `\FORMAT{`⟨*format name*⟩`}`, and any packages or classes loaded can be indicated with

`\CLASS[`⟨*options*⟩`]{`⟨*class name, version*⟩`}`
`\PACKAGE[`⟨*options*⟩`]{`⟨*package name, version*⟩`}`

These do not provide information that is useful for automated checking; after all, packages change their version numbers frequently. Rather, including this information in a test indicates the conditions under which the test was definitely known to pass at a certain time in the past.

The `\END` command signals the end of the test (but read on). Some additional diagnostic information is printed at this time to debug if the test did not complete 'properly' in terms of mismatched brace groups or `\if`...`\fi` groups.

In a LaTeX document, `\end{document}` will implicitly call `\END` at the very end of the compilation process. If `\END` is used directly (replacing `\end{document}` in the test), the compilation will halt almost immediately, and various tasks that `\end{document}` usually performs will not occur (such as potentially writing to the various `.toc` files, and so on). This can be an advantage if there is additional material printed to the log file in this stage that you wish to ignore, but it is a disadvantage if the test relies on various auxiliary data for a subsequent typesetting run. (See the `checkruns` variable for how these tests would be test up.)

## 2.2 Commands to help write tests

A simple command `\CHECKCOMMAND\⟨macro⟩` is provided to check whether a particular `\⟨macro⟩` is defined, undefined, or equivalent to `\relax`. This is useful to flag either that internal macros are remaining local to their definitions, or that defined commands definitely are defined, or even as a reminder that commands you intend to define in a future package need to be tested once they appear.

`\TYPE` is used to write material to the `.log` file, like LaTeX's `\typeout`, but it allows 'long' input. The following commands are defined to use `\TYPE` to output strings to the `.log` file.

- `\SEPARATOR` inserts a long line of `=` symbols to break up the log output.
- `\NEWLINE` inserts a linebreak into the log file.
- `\TRUE`, `\FALSE`, `\YES`, `\NO` output those strings to the log file.
- `\ERROR` is *not* defined but is commonly used to indicate a code path that should never be reached.
- The `\TEST{`⟨*title*⟩`}{`⟨*contents*⟩`}` command surrounds its ⟨*contents*⟩ with some `\SEPARATOR`s and a ⟨*title*⟩.
- `\TESTEXP` surrounds its contents with `\TYPE` and formatting to match `\TEST`; this can be used as a shorthand to test expandable commands.
- TODO: would a `\TESTFEXP` command (based on `\romannumeral` expansion) be useful as well?

An example of some of these commands is shown following.

```
\TEST{bool_set,~lazy~evaluation}
 {
  \bool_set:Nn \l_tmpa_bool
   {
    \int_compare_p:nNn 1=1
    && \bool_if_p:n
     {
      \int_compare_p:nNn 2=3 ||
      \int_compare_p:nNn 4=4 ||
```

```
      \int_compare_p:nNn 1=\ERROR % is skipped
     }
    && \int_compare_p:nNn 2=2
   }
  \bool_if:NTF \l_tmpa_bool \TRUE \FALSE
 }
```

This test will produce the following in the output.

```
============================================
TEST 8: bool_set, lazy evaluation
============================================
TRUE
============================================
```

(Only if it's the eighth test in the file of course, and assuming expl3 coding conventions are active.)

## 2.3 Showing box content

The commands introduced above are only useful for checking algorithmic or logical correctness. Many packages should be tested based on their typeset output instead; TeX provides a mechanism for this by printing the contents of a box to the log file. The `regression-test.tex` driver file sets up the relevant TeX parameters to produce as much output as possible when showing box output.

A plain TeX example of showing box content follows.

```
\input regression-test.tex\relax
\START
\setbox0=\hbox{\rm hello \it world $a=b+c$}
\showbox0
\END
```

This produces the output shown in Figure 5 (left side). It is clear that if the definitions used to typeset the material in the box changes, the log output will differ and the test will no longer pass.

The equivalent test in LaTeX 2$_\varepsilon$ using expl3 is similar.

```
\input{regression-test.tex}
\documentclass{article}
\usepackage{expl3}
\START
\ExplSyntaxOn
\box_new:N \l_tmp_box
\hbox_set:Nn \l_tmp_box {hello~ \emph{world}~ $a=b+c$}
\box_show:N \l_tmp_box
\ExplSyntaxOff
\END
```

```
> \box0=
\hbox(6.94444+0.83333)x90.56589
.\tenrm h
.\tenrm e
.\tenrm l
.\tenrm l
.\tenrm o
.\glue 3.33333 plus 1.66666 minus 1.11111
.\tenit w
.\tenit o
.\tenit r
.\tenit l
.\tenit d

.\glue 3.57774 plus 1.53333 minus 1.0222
.\mathon
.\teni a
.\glue(\thickmuskip) 2.77771 plus 2.77771
.\tenrm =
.\glue(\thickmuskip) 2.77771 plus 2.77771
.\teni b
.\glue(\medmuskip) 2.22217 plus 1.11108 minus 2.22217
.\tenrm +
.\glue(\medmuskip) 2.22217 plus 1.11108 minus 2.22217
.\teni c
.\mathoff

! OK.
l.9 \showbox0
```

```
> \box71=
\hbox(6.94444+0.83333)x91.35481
.\OT1/cmr/m/n/10 h
.\OT1/cmr/m/n/10 e
.\OT1/cmr/m/n/10 l
.\OT1/cmr/m/n/10 l
.\OT1/cmr/m/n/10 o
.\glue 3.33333 plus 1.66666 minus 1.11111
.\OT1/cmr/m/it/10 w
.\OT1/cmr/m/it/10 o
.\OT1/cmr/m/it/10 r
.\OT1/cmr/m/it/10 l
.\OT1/cmr/m/it/10 d
.\kern 1.03334
.\glue 3.33333 plus 1.66666 minus 1.11111
.\mathon
.\OML/cmm/m/it/10 a
.\glue(\thickmuskip) 2.77771 plus 2.77771
.\OT1/cmr/m/n/10 =
.\glue(\thickmuskip) 2.77771 plus 2.77771
.\OML/cmm/m/it/10 b
.\glue(\medmuskip) 2.22217 plus 1.11108 minus 2.22217
.\OT1/cmr/m/n/10 +
.\glue(\medmuskip) 2.22217 plus 1.11108 minus 2.22217
.\OML/cmm/m/it/10 c
.\mathoff

! OK.
<argument> \l_tmp_box

l.12 \box_show:N \l_tmp_box
```

Figure 5: Output from displaying the contents of a simple box to the log file, using plain TEX (left) and expl3 (right). Some blank lines have been added to the plain TEX version to help with the comparison.

The output from this test is shown in Figure 5 (right side). There is marginal difference (mostly related to font selection and different logging settings in LATEX) between the plain and expl3 versions.

When examples are not self-contained enough to be typeset into boxes, it is possible to ask TEX to output the entire contents of a page. Insert `\showoutput` for LATEX or set `\tracingoutput` positive for plain TEX; ensure that the test ends with `\newpage` or equivalent because TEX waits until the entire page is finished before outputting it.

TODO: should we add something like `\TRACEPAGES` to be format-agnostic here? Should this perhaps even be active by default?

```
\input regression-test.tex\relax
\START
\TEST{counter-math}{
%<*test>
  \OMIT
  \newcounter{numbers}
  \setcounter{numbers}{2}
  \addtocounter{numbers}{2}
  \stepcounter{numbers}
  \TIMO
  \typeout{\arabic{numbers}}
%</test>
%<expect>  \typeout{5}
}
\END
```

Figure 6: Test and expectation can be specified side-by-side in a single `.dtx` file.

```
 \generate{\file{\jobname.lvt}{\from{\jobname.dtx}{test}}
           \file{\jobname.lve}{\from{\jobname.dtx}{expect}}}
```

Figure 7: Test and expectation are generated from a `.dtx` file of the same name.

# 3 Alternative test formats

## 3.1 Generating test files with DocStrip

It is possible to pack tests inside source files. Tests generated during the unpacking process will be available to the `check` and `save` commands as if they were stored in the `testfiledir`. Any explicit test files inside `testfiledir` take priority over generated ones with the same names.

## 3.2 Specifying expectations

Regression tests check whether changes introduced in the code modify the test output. Especially while developing a complex package there is not yet a baseline to save a test goal with. It might then be easier to formulate the expected effects and outputs of tests directly. To achieve this, you may create an `.lve` instead of a `.tlg` file.[1] It is processed exactly like the `.lvt` to generate the expected outcome. The test fails when both differ.

Combining both features enables contrasting the test with its expected outcome in a compact format. Listing 6 exemplary tests TeXs counters. Listing 7 shows the relevant part of an `.ins` file to generate it.

[1]Mnemonic: `lvt`: **t**est, `lve`: **e**xpectation

```
function setversion_update_line(line, date, version)
  local i
  -- No real regex so do it one type at a time
  for _,i in pairs({"Class", "File", "Package"}) do
    if string.match(
      line,
      "^\\Provides" .. i .. "{[a-zA-Z0-9%-]+}%[[^%]]*%]$"
    ) then
      line = string.gsub(line, "%[%d%d%d%d/%d%d/%d%d", "["
        .. string.gsub(date, "%-", "/")
      line = string.gsub(
        line, "(%[%d%d%d%d/%d%d/%d%d)␣[^␣]*", "%1␣" .. version
      )
      break
    end
  end
  return line
end
```

Figure 8: Example `setversion_update_line` function.

# 4 Release-focussed features

## 4.1 Automatic version modification

As detailed above, the `setversion` target will automatically edit source files to modify date and version. This behaviour is governed by variable `""`. As standard, no automatic replacement takes place, but setting `""` will allow this to happen, with options

- `ProvidesPackage` Searches for lines using the LaTeX 2ε `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile` identifiers (as a whole line).
- `ProvidesExplPackage` Searches for lines using the expl3 `\ProvidesExplPackage`, `\ProvidesExplClass` and `\ProvidesExplFile` identifiers (at the start of a line).
- `filename` Searches for lines using `\def\filename`, `\def\filedate`, ..., formulation.
- `ExplFileName` Searches for lines using `\def\ExplFileName`, `\def\ExplFileDate`, ..., formulation.

For more complex cases, the programmer may directly define the Lua function `setversion_update_line()`, which takes as arguments the line of the source, the supplied date and the supplied version. It should return a (possibly unmodified) line and may use one, both or neither of the date and version to update the line. Typically, `setversion_update_line` should match to the exact pattern used by the programmer in the source files. For example, for code using macros for the date and version a suitable function might read as shown in Figure 8.

## 4.2 Typesetting documentation

As part of the overall build process, l3build will create PDF documentation as described earlier. The standard build process for PDFs will attempt to run Biber, BibTeX and MakeIndex as appropriate (the exact binaries used are defined by `"biber"`, `"bibtex8"` and `"makeindex"`). However, there is no attempt to create an entire PDF creation system in the style of `latexmk` or similar.

For package authors who have more complex requirements than those covered by the standard set up, the Lua script offers the possibility for customisation. The Lua function `typeset` may be defined before reading `l3build.lua` and should take one argument, the name of the file to be typeset. Within this function, the auxiliary Lua functions `biber`, `bibtex`, `makeindex` and `tex` can be used, along with custom code, to define a PDF typesetting pathway. The functions `biber` and `bibtex` take a single argument: the name of the file to work with *minus* any extension. The `tex` takes as an arugment the full name of the file. The most complex function `makeindex` requires the name, input extension, putput extension, log extension and style name. For example, Figure 9 shows a simple script which might apply to a case where multiple BibTeX runs are needed (perhaps where citations can appear within other references).

```
#!/usr/bin/env texlua

-- Build script with custom PDF route

module = "mymodule"

function typeset (file)
  local name = string.match (file, "^(.*)%.") or name
  local errorlevel = tex (file)
  if errorlevel == 0 then
    -- Return a non-zero errorlevel if anything goes wrong
    errorlevel = (
      bibtex (name) +
      tex (file)    +
      bibtex (name) +
      tex (file)    +
      tex (file)
    )
  end
  return errorlevel
end

kpse.set_program_name("kpsewhich")
dofile(kpse.lookup("l3build.lua"))
```

Figure 9: A customised PDF creation script.

# Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.


**Symbols**

\⟨*macro*⟩ . . . . . 13, 13

**A**

\AUTHOR . . . . . 13

**C**

\CHECKCOMMAND . . . . . 13
\CLASS . . . . . 13

**E**

\END . . . . . 11
\ERROR . . . . . 14

**F**

\FALSE . . . . . 14
\fi . . . . . 13
\FORMAT . . . . . 13

**I**

\if . . . . . 13

**N**

\NEWLINE . . . . . 14
\newpage . . . . . 15
\NO . . . . . 14

**O**

\OMIT . . . . . 11
\openin . . . . . 12
\openout . . . . . 12

**P**

\PACKAGE . . . . . 13
\ProvidesClass . . . . . 18
\ProvidesExplClass . . . . . 18
\ProvidesExplFile . . . . . 18
\ProvidesExplPackage . . . . . 18
\ProvidesFile . . . . . 18
\ProvidesPackage . . . . . 18

**R**

\relax . . . . . 13
\romannumeral . . . . . 14

**S**

\SEPARATOR . . . . . 14, 14
\showoutput . . . . . 15
\START . . . . . 11

**T**

\TEST . . . . . 14, 14
\TESTEXP . . . . . 14
\TESTFEXP . . . . . 14
\TIMO . . . . . 11
\TRACEPAGES . . . . . 15
\tracingoutput . . . . . 15
\TRUE . . . . . 14
\TYPE . . . . . 14, 14, 14
\typeout . . . . . 14

**Y**

\YES . . . . . 14